Automated Measurement of Models of Requirements

Martin Monperrus, Benoit Baudry, Joël Champeau, Brigitte Hoeltzener, Jean-Marc Jézéquel

Automated Measurement of Models of Requirements

Martin Monperrus[1], Benoit Baudry[2], Joël Champeau[3],
Brigitte Hoeltzener[3], Jean-Marc Jézéquel[2,4]
1. TU Darmstadt - 2. INRIA - 3. ENSIETA - 4. University of Rennes 1

Abstract

On way to formalize system requirements is to express them using the object-oriented paradigm. In this case, the class model representing the structure of requirements is called a requirements metamodel, and requirements themselves are object-based models of natural-language requirements. In this paper, we show that such object-oriented requirements are well-suited to support a large class of requirements metrics. We define a requirements metamodel and use an automated measurement approach proposed in our previous work to specify requirements metrics. We show that it is possible to integrate 78 metrics from 11 different papers in the proposed framework. The software that computes the requirements metric values is fully generated from the specification of metrics.

1 Introduction

Requirements Metrics - Value Added. This was an appealing title of a talk at the 1997 International Conference on Requirements Engineering [1]. The idea behind the title is that it is possible to identify risks and flaws very early in the system life cycle by measuring requirements.

Also, the Capability Maturity Model (CMM) of the Software Engineering Institute emphasizes on the need for requirements measurements: *measurements are made and used to determine the status of the activities for managing the allocated requirements* (*Key Practices of the Capability Maturity Model* [2, p. 98]). These guidelines on requirements measurement have then been extended and refined in the ISO 25000 series of standards (SQuaRE - [3]).

According to Zave [4], requirements measurement is an important subfield of research on requirements engineering, as a means of comparing solutions to requirements engineering problems. Indeed, several papers have defined requirements metrics (e.g. [5, 6, 7, 8, 9, 10, 11, 12]).

However, those papers all address different aspects of requirements metrics, e.g. measuring the product (such as counting the number of words of the requirements specification), or measuring the process (such as gathering the cost in man/month of the requirements engineering phase). Also, certain metrics do not share the same terminology, i.e. are described using different terms. For instance, the notions of *time frame* and *unit of time* may have similar yet different meanings.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

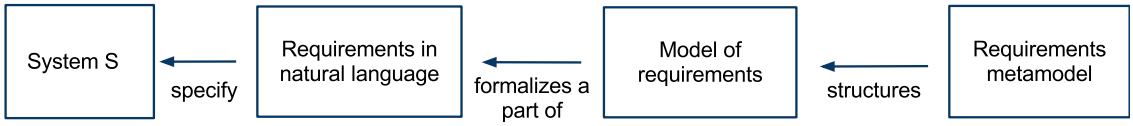

Figure 1: Our acceptation of *model of requirements*.

Finally, some metrics are ambiguous and subject to interpretation since they are all described with natural language. For instance, the *cost of change to requirements* of [8] could be interpreted as the cost of verifying the consistency of impacted requirements or the cost of modifying the corresponding software items. To our knowledge, there is no related work trying to unify previous approaches in a single framework.

On the other hand, in recent years, a body of techniques called *model-driven engineering* has emerged to tackle several problems of software engineering [13]. A key insight of model-driven engineering is to leverage the object-oriented paradigm in other areas than domain analysis or implementation. For instance, previous work [14] showed that the object-oriented paradigm is appropriate to describe requirements specification using *models of requirements*. The research presented in this paper goes further in this direction and explores the confluence of requirements measurement and model-driven engineering.

Figure 1 illustrates what we call a *model of requirements*. A standard view of requirements engineering in that a set of requirements written in natural languages specifies a system S (at the left-hand side of the figure). In this paper, along the same line as related research [15], we define a model of requirements as a formalization of a part of a requirements specification using the object-oriented paradigm, i.e. certain requirements are described as objects having fields and relations with other objects. This model of requirements is an instance of a requirements metamodel that defines the space of valid models of requirements (at the right-hand side of the figure). Since a requirements metamodel describes the structure of objects representing requirements, it is similar to a set of classes of an object-oriented program and can be represented using a class diagram. Thus a model of requirements as used in this paper, is different from a requirements process model that describes a customer process for better understanding the environment, and also different from an analysis model for showing how a delivered product or solution will work in the client environment.

In this paper, we propose a requirements metamodel that contains all the necessary concepts with respect to requirements measurement, i.e. a requirements metamodel that supports the requirements metrics of the literature. The metamodel unifies both the terminology and the semantics of the requirements concepts involved. More specifically, our contributions are as follows:

- a unified list of 78 requirements metrics based on previous work on requirements metrics (Section 2). This list grounds the contributions that follow but may also be used in the context of new requirements quality models.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

- a requirements metamodel that supports requirements measurement (Section 4). The full specification of this metamodel is given in appendix A.
- the automatic generation of a requirements measurement tool using the model-driven measurement approach of [16]. The aforementioned unified requirements metrics are formally captured as instances of a metric metamodel. These instances then seed the generation of a requirements measurement tool (Section 5).

We analyze the main characteristics of our approach in a discussion section, where we provide insights and arguments showing that: 1) it unifies previous heterogeneous work; 2) it allows the complete computability of requirements metrics; 3) it is fully supported by generative programming.

The remainder of this paper is organized as follows. Section 2 is the study of the literature on requirements metrics; Section 3 presents the process we adopted to build our requirements metamodel, which is presented in section 4. Then, section 5 describes our model-driven solution to the measurement of requirements. Finally, section 6 discusses our approach and section 7 concludes the paper by sketching future work.

2 A Survey on Requirement Metrics

In this section, we present a survey on requirements metrics. First we discuss the criteria we chose to select metrics among previous contributions. We then briefly describe the corresponding papers.

2.1 Methodology

2.1.1 Identification of papers

To identify the relevant papers for the survey, we have used the Google Scholar[1] bibliographic database with the words "requirement" and "metric" or "measurement". Furthermore, we have carefully followed the citation graph to ensure that we have not missed papers that are not indexed in Google Scholar. All matching papers were systematically considered for inclusion.

They are two threats to validity in this identification strategy. First, Google Scholar may not index all papers related to requirements metrics. Indeed, Zhang and Ali Babar [17] showed that IEEE Xplore and ACM Digital Library are the most commonly used sources for literature reviews in software engineering. However, several authors (e.g. [18, 19]) showed that the coverage of Google Scholar is very good in average (close to 100% according to the latest paper), independently of the field. Second, it is possible that we have missed relevant papers which do not mention the

[1] `http://scholar.google.com`

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

aforementioned words. We think it is unlikely since these words are both used in papers on requirements measurement that are much cited (e.g. [6]) or written by reference authors (e.g. [4]).

2.1.2 Selection of requirements metrics

We have defined the following criteria to select metrics among existing papers.

1. Our work focuses on requirements product metrics. In this paper, a requirement product metric is a metric that is applied to a requirement or a set of requirements, also known as a requirements specification [20].

2. We discard purely syntactic and natural language based metrics, for instance the *number of pages* of [6]. Syntactic metrics are useful, and they perfectly address the measurement of existing requirements specifications. However, they are not appropriate to obtain semantic information with syntactic metrics (for instance, the number of requirements that were discarded between the first and the second version of the requirements specification). Although natural language based metrics also perfectly fit to existing requirements specifications (i.e. they are lightweight), they are often imprecise [21].

2.2 Selected papers

In the following, we briefly present pieces of research that define requirements metrics which comply with the criteria defined in section 2.1.2. They are mostly sorted by chronological order.

Baumert et al. The paper of Baumert et al. [22] describes a set of software measures that are compatible with the measurement practices of the Capability Maturity Model for Software. The measures are classified by category. For instance, one category addresses the requirements stability.

Davis et al. The goal of the Davis et al.'s paper [5] is to thoroughly explore the concept of quality in a software requirements specification (SRS) and to define quality attributes that can be really measured. They define 24 quality attributes for a software requirements specification. They show examples of requirements that satisfy or not each quality attribute.

Costello and Liu Costello and Liu [6] believe that the discipline of software metrics can be applied to requirements metrics. Indeed, they try to provide a full life cycle coverage by metrics. Their final goal is to comprehensively assess objective aspects of the requirement engineering processes and products. To our knowledge, they are the first to introduce the expression "measurable requirements specification". This expression emphasizes the key role of measurement for requirement engineering. To a certain extent, this means that the will to measure requirements is a sufficient reason to modify the requirement products or the

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

requirement engineering process accordingly. Costello and Liu define several metrics linked to three quality attributes: volatility, traceability and completeness.

Marchesi To the best of our knowledge, the paper by Marchesi [7] is the first to consider requirements from a model-driven viewpoint. The corresponding metrics address use case based requirements, in the sense of UML use cases [23].

Loconsole Loconsole [8] also defines a set of requirement products metrics. In this paper, she applies the Goal/Question/Metric approach [24] to the Capability Maturity Model (similarly to [22]) and obtains 53 interesting requirements metrics. According to the criteria of section 2.1.2, we keep 13 of them.

Henderson-Sellers et al. The paper by Henderson-Sellers et al. [9] makes a synthesis between the objections of Costello [6] and the idea of Marchesi [7]. Henderson-Sellers et al. set out a use case template *so that use cases can be metricated*. Given a set of requirements expressed in a standard use cases template, it is then possible to define requirements metrics and obtain metric values. Indeed, they propose twelve use case metrics.

Singh et al. Singh et al. define [10] a complexity metric for an individual requirement and for a category of requirements. In order to compute the metric values, they propose a requirements metamodel. To our knowledge, this is the first attempt to define a requirements metamodel with the main goal of making the requirements measurable.

White papers In 2004, two white papers by different companies were published [25], [26]. Kolde points out that many projects lack requirements measurement and that requirements documents are of various form, hence are difficult to measure. He also defines several requirements metrics. The scope of *Computing Model Complexity* [26] is larger. However, since the company sells a UML-oriented tool, the paper contains several use cases metric definitions. Both papers emphasize on the fact that requirements metamodeling eases the measurement of requirements.

MDD Engineering Metrics Catalogue The Modelware project published a *MDD Engineering Metrics Catalogue* [27]. We include in our approach the metrics related to use cases that are defined in this document.

Berenbach et al. Berenbach et al. [11] describes a CMMI compliant and model-driven approach for requirements measurement. This work maps the CMM process onto models, mainly use cases models, in order to automatically obtain metric values.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

#	Short Metric Description	Origins
1	Number of requirements (NR)	Costello, Kolde, Davis, Loconsole, Baumert
2	Number of initial requirements	Loconsole, Baumert
3	Number of requirements added per time frame	Costello, Kolde
4	Number of requirements modified per time frame	Costello
5	Number of requirements deleted per time frame	Costello, Baumert
6	Number of changes per time frame	Costello, Kolde, Loconsole
7	Number of changes per requirement	Loconsole
8	Number of requirements that trace to the next level up	Costello
9	Number of requirements that trace to the next level down	Costello
10	Number of requirements that trace to the next level in both directions	Costello
11	Number of requirements that trace from highest to lowest	Costello, Kolde
12	Number of requirements that trace from lowest to highest	Costello
13	Number of CSCI linked to a requirement	Loconsole
14	Number of requirements per level that have inconsistent traceability links upward	Costello
15	Number of req. per level that have inconsistent traceability links downward	Costello, Kolde
16	Number of requirements per level that have no traceability links upward	Costello
17	Number of requirements per level that have no traceability links downward	Costello
18	Degree of decomposition per requirement per time frame	Costello
19	Number of requirements per status	Costello, Kolde, Davis, Berenbach, Loconsole
20	Number of req. that trace to one or more incomplete req.	Costello
21	Num. of req. that trace to inconsistent requirement (i.e. status is Tbx - to be X)	Costello
22	Number of incomplete requirements	Costello
23	Number of requirements reflected in one or more CSCI	Costello
24	Number of use cases per status	Modelware
25	Number of use cases per status per time frame	Modelware, Loconsole
26	Number of use cases	Douglass, Marchesi
27	Number of functions specified (NF)	Davis
28	Number of unique functions specified (NUF)	Davis
29	Number of requirements traced to incomplete CSCI	Costello
30	Number of accepted use case diagrams	Berenbach
31	Number of non submitted use case diagrams	Berenbach
32	Number of sequence diagrams per use case	Douglass
33	Number of submitted use case diagrams	Berenbach
34	Number of boundaries that do not communicate with an actor	Berenbach
35	Number of boundaries that do not communicate with a concrete use case	Berenbach
36	Number of use cases per actor	Douglass, Marchesi
37	Number of actors	Berenbach, Douglass, Henderson-Sellers
38	Number of use cases non described by one or more behavioral diagram	Berenbach
39	Number of use cases that do not appear on a diagram	Berenbach
40	Number of circular dependencies between use cases	Berenbach
41	Number of uses cases that do not appear on a parent behavioral diagram	Berenbach
42	Number of mixed use cases (including one abstract and one concrete)	Berenbach
43	Number of impacted requirements per change	Modelware, Loconsole, Baumert
44	Number of input states per function (A)	Davis
45	Number of states per use cases	Douglass
46	Number of activities per use cases	Douglass, Henderson-Sellers
47	Number of activities in the main flow per use case	Henderson-Sellers
48	Number of activities per alternative flow per use case	Henderson-Sellers
49	Number of activities in the alternative flows per use case	Henderson-Sellers
50	Number of activities per actor	Henderson-Sellers, Marchesi
51	Number of activities per goal	Henderson-Sellers
52	Number of goals per stakeholder	Henderson-Sellers
53	Number of dependencies per use case (includes, extends)	Douglass
54	Number of requirements changes to a requirements baseline	Kolde
55	Number of requirements by responsible	Kolde
56	Number of responsibles by requirement	Loconsole
57	Number of functional requirements allocated to a project release	Kolde, Loconsole, Baumert
58	Strength of an individual requirement	Singh
59	Strength of a category	Singh
60	Number of req. for which all reviewers presented identical interpretations (NU)	Davis, Loconsole
62	Number of input stimulus per function (B)	Davis
63	Number of flows per function (C)	Davis
65	Number of correct requirements (NC)	Davis
67	Verifiability = $NR/(NR + \sum_i cost_i + \sum_i time_i)$	Davis
68	Number of test cases per requirement	Loconsole, Baumert
69	Number of fun. that are not deterministic (NUFND)	Davis
70	Number of req. that describe pure external behavior	Davis
71	Number of req. that describe architecture and algorithm (NAC)	Davis
74	Size of the longest path between the first activity and the final activity	Henderson-Sellers
75	Number of alternative flows	Henderson-Sellers
76	Number of stakeholders	Henderson-Sellers
77	Number of goals	Henderson-Sellers
78	Number of changes to req. incorporated into baseline per time frame	Loconsole
	DERIVED	
61	Unambiguity (derived)	Davis
64	Completeness per function (derived)	Davis
66	Correctness (derived)	Davis
72	Design dependency (derived)	Davis
73	Redundancy (derived)	Davis

Table 1: Consolidated List of Requirements Metrics From Literature

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

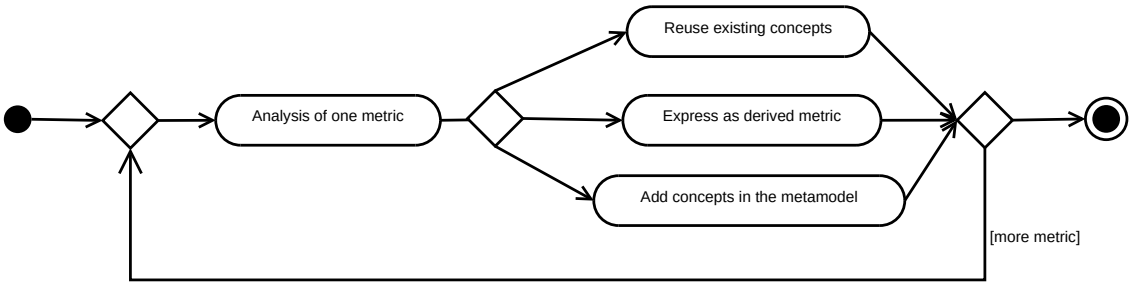

Figure 2: Our metric-driven metamodeling process

2.3 A Comprehensive List of Requirements Metrics

The unification of the metrics presented in section 2.2 results in a list of 78 metrics that is shown in Table 1. The first column of the table is an ID, the second column gives a short metric description and the last column contains the name of the first author of papers that describe a particular metric. Each row represents a metric. Note that for certain metrics, we had to unify terms, i.e. certain metrics of the literature deal with concepts that are similar yet named differently. In the following, we will use this list in two manners:

- First, to create a requirements metamodel dedicated to measurement.
- Second, to formally implement these metrics using the model-driven measurement approach that we have presented in [16].

We also hope that this unified list will inspire measurement features in both commercial and open-source requirements tools and will contribute to ground new requirements quality models (e.g. [28]).

3 A Metric-driven Metamodeling Process

This section presents a metric-driven metamodeling process. It is notable that only the need for measurement triggers the metamodeling activity (i.e. it's not the simple reuse of an existing metamodel). Hence, we use the term "metric-driven", which means that:

- the main goal of the metamodel is to support the specification and implementation of requirements metrics.
- the metamodel is created with a bottom-up approach. Every concept of the metamodel (class, reference, etc.) has been created because it was needed in a particular requirements metric of the literature.

Figure 2 shows the process we followed to create the requirements metamodel. It is a UML activity diagram. At the beginning, the requirements metamodel contains nothing. Then, we

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

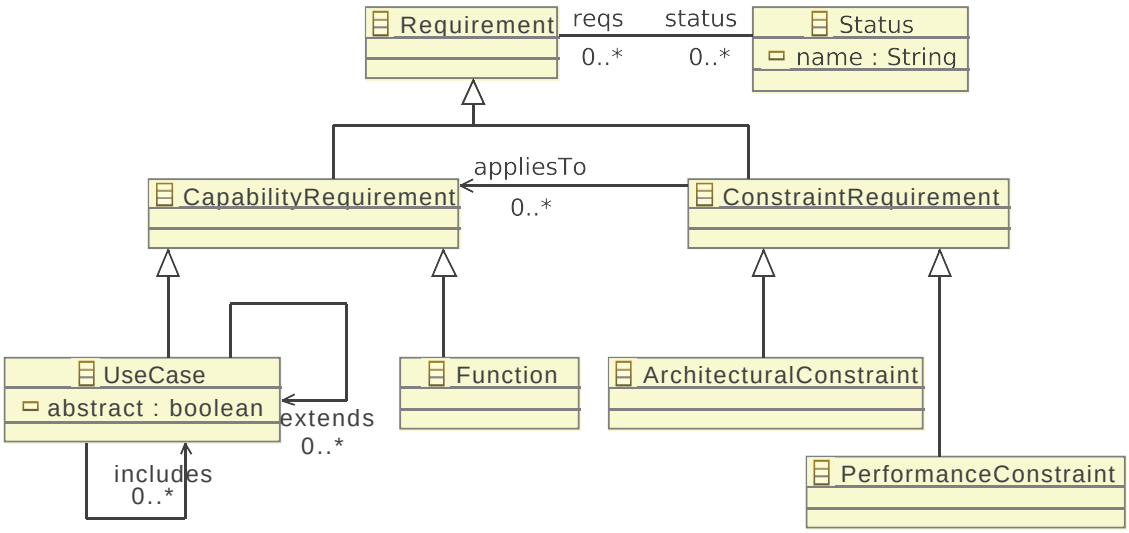

Figure 3: The requirements inheritance hierarchy

analyzed each requirement metric one by one: if the metric referred to concepts or relationships that do not yet exist in the requirements metamodel at this point in time, we added them to the metamodel (obviously, the first considered metric triggered at least one concept to the metamodel, for instance the class "Requirement"). Also some metrics can be expressed as derived metrics, in such cases, we simply defined them on top of the already analyzed metrics.

The analysis process was solely done by the first author in 2 days. Unfortunately, we did not trace all the detail of this work. Hence, we can not produce the number of metrics that could be expressed directly, or that triggered the addition of a new concept in the requirements metamodel.

Eventually the application of our metric-driven metamodeling process for requirements engineering ended up with a metamodel containing 36 classes. This small number of classes shows that the majority of metrics deal with the same requirements concepts. In other terms, not all analyses of a particular metric triggered the addition of a new concept. The core of the requirements metamodel was identified after having analyzed the first half of metrics. Eventually, we obtained a metamodel that captures the common requirements modeling concepts and the relationships between these concepts. This metamodel is described in the next section.

4 A Requirements Metamodel that Supports Measurement

In this section, we present the main aspects of our requirements metamodel that supports all the requirements metrics of the literature. Not all classes, references and attributes are presented here but the complete metamodel is given in appendix.

The metamodel is centered on the notion of requirement, as shown in Figure 3. A requirement can be refined in several types. A *CapabilityRequirement* specifies an atomic capability of the system. It can be defined using textual description with the class *Function*, or using a *UseCase*. The

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

main difference between *UseCase* and *Function* is that use cases involve a concrete scenario and some actors. On the contrary, a requirement instance of *Function* is an abstract description of the capability. In our approach, since *UseCase* inherits from *CapabilityRequirement*, we recommend to have a 1-to-1 mapping between uses cases and capabilities and to extensively use the "includes" relationship between use cases to handle higher-order use-cases and complex scenarios.

Some requirements are expressed as constraints on the system (also known in the literature as non-functional requirements, but we keep the terminology of Davis et al. [5]), expressed with the *ConstraintRequirement* class, which can be refined as *ArchitecturalConstraint* or as *PerformanceConstraint*. An *ArchitecturalConstraint* represents the required constraints on the system architecture (for example, *"the messaging system has to run on Unix"*). A *PerformanceConstraint* represents an expectation on the performance of the system (for example, *"the system has to respond in less than 500ms"*). The metamodel supports links from a particular constraint (instance of *ConstraintRequirement*) to a capability: this is handled by the reference between *ConstraintRequirement* and *CapabilityRequirement*. For instance, *"the messaging system has to run on Unix"* may be linked to the capability *"a message is added when new users are added."*. Note that *ConstraintRequirement* is not abstract, which means that one can instantiate this class directly to express other constraints, for example security or reliability requirements.

These classes of the metamodel are all derived from requirements metrics that target a specific requirement type. For instance, the metric *number of req. that describe architecture and algorithm* [5] counts the number of instances of the *ArchitecturalConstraint* class.

The main attributes and references of the class *Requirements* are shown in Figure 4. The requirement itself has a name and is described as one string containing few sentences. Note that this requirement metamodel does not address the finer grain modeling of a requirement. A creation date is a time stamp for traceability. Several metrics are concerned with the history of a requirement, hence a requirement can be tagged current version, while keeping a traceability link to older versions thanks to the reference *pastVersions*. A requirement is associated with a *status*, instance of class *Status*. The different statuses are not coded in the metamodel, but as a library of instances of class *Status* dependent of the process, for instance, *to be submitted* (TBS), *to be approved* (TBA), *approved* (A). The requirements can be structured into categories, that is why there is a *category* attribute. Categories are logical packages, in order to facilitate the comprehension of the requirements specifications. For instance, a category may correspond to a macro-function of the system, e.g.; "Entertainment system" in an airplane. A requirement can be associated to zero or more *TestCase*s. The modeling of test cases is not is the scope of this metamodel. A requirement is finally allocated to zero or more software items (also known as *Computer Software Configuration Item* - CSCI). The remaining associations express the decomposition in sub-requirements and the dependency links between requirements. The latter is equivalent to the dependency matrices of previous approaches [29].

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

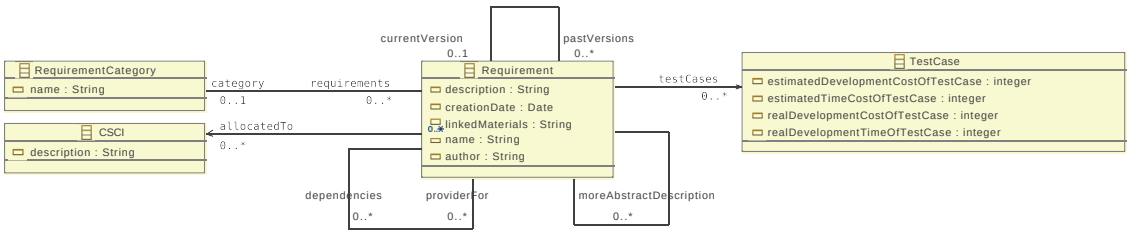

Figure 4: The requirement concept

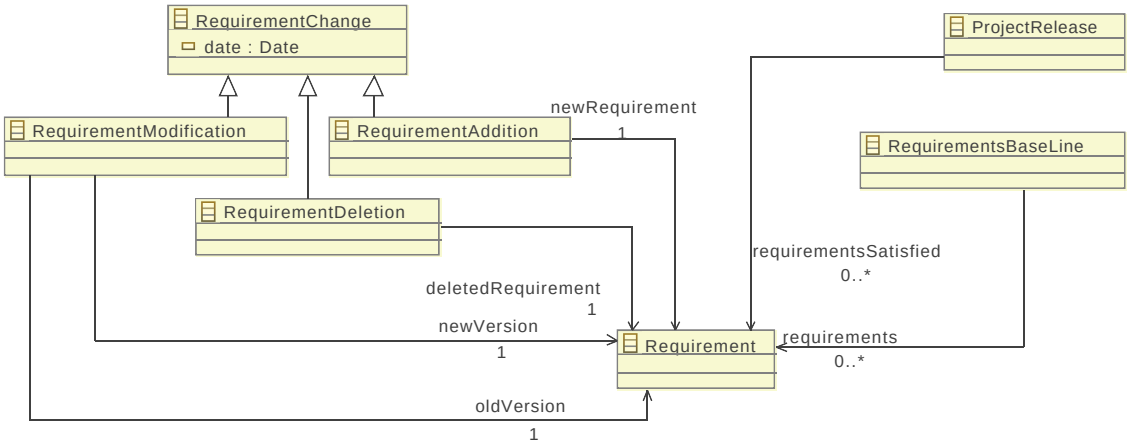

Figure 5: Life cycle management of requirements

Figure 5 shows the concepts linked to the life cycle of requirements. This family of concepts is part of an important number of requirements metrics of the survey. A *Baseline* is composed of a set of well formed requirements. A *Release* satisfies a set of requirements. Within a time frame, whose meaning depends on the requirements process, there are several *RequirementChange*. *RequirementChange* is an abstract class, specialized into *RequirementAddition*, *RequirementModification*, and *RequirementDeletion*. These classes allow a full control over the requirements life cycle.

Apart from these main concepts, our requirements metamodel contains many more concepts that will be mentioned in the next sections. The full metamodel is given in appendix.

5 Implementation of Requirements Measurement Software Using Model-driven Development

To implement the metrics listed in Table 1, we have used the model-driven measurement approach (MDM) presented in [16, 30]. It means that we have expressed the metrics of Table 1 as an instance of a metric metamodel. These formal metrics refer to the metric metamodel described above. Figure 6 presents the model-driven measurement approach in the context of requirements measurement as an UML activity diagram. The application of the MDM approach begins by cre-

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

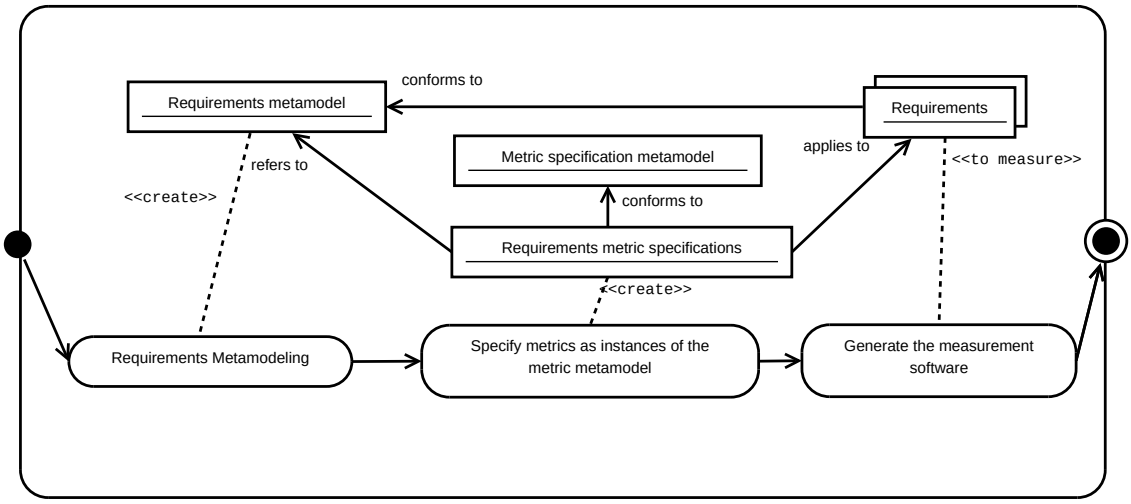

Figure 6: Model-driven measurement of requirements

ating a domain metamodel: in our case the requirements metamodel presented above. Then, the MDM approach consists in specifying metrics as instances of a metric specification metamodel[2]. Note that the metric metamodel is domain-independent. It only contains concepts related to metrics. Eventually, a prototype that implements the MDM approach fully generates the measurement software based on the metric specifications.

Figure 6 contains not only activities, but also the artifacts that are involved in the interplay of the activities (represented by UML objects, i.e. rectangles with underlined text). A set of requirements metric specifications is an instance of the metric specification metamodel and refers to concepts of the requirements metamodel. There is no dependency between the metric metamodel which is independent of the requirements domain, and the requirements metamodel, which is independent of the MDM approach. What is generated is a tool for measuring requirements specifications. The generated measurement tool is dedicated to the metrication of models of requirements structured by the requirements metamodel defined above. This tool is fully fledged, and once deployed, can be used by requirements engineers and managers to get quantitative feedback on the requirements they work on.

Let us now elaborate on a sample of three metrics formally specified with the MDM approach.

5.1 Example 1: Total number of requirements

In MDM, a *SigmaMetric* counts the number of model elements satisfying a predicate. The total number of requirements is expressed as a *SigmaMetric*. Note that since our requirements metamodel handles the version history, we have to select only current requirements. If self.currentVersion is set, it means that the requirement has been overridden by a new one, hence we count only those which are not linked to a current version.

[2]Please refer to [30] for details about the metric specification metamodel, which is out of scope of this paper.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

```
metric SigmaMetric 01_NOR is
elements satisfy "(Requirement.isInstance(self)
and self.currentVersion==void)"
endmetric
```

5.2 Example 2: Number of CSCI linked to a requirement

In MDM, a *SetOfElementsPerX* metric counts the number of elements linked to a root element by a path of a certain kind. Hence, this is specified using three predicates: 1) a predicate on the root element; 2) a predicate on the counted element, 3) a predicate on the path, which lists the references that can be followed. Note that if they are several elements matching as root element, we obtain one metric value per root element. The number of CSCI linked to a requirement counts the number of CSCI linked to a requirement by references of type *refinedIn* or *allocatedTo*.

```
metric SetOfElementsPerX 13_N is
elements satisfy "Requirement.isInstance(self)"
count "CSCI.isInstance(self)"
references followed "refinedIn,allocatedTo"
endmetric
```

5.3 Example 3: Degree of decomposition per requirement

In MDM, a *PathLength* gives the size of the longest path from a root element following a certain path. It is specified using two predicates: 1) on the root element and 2) on the path. The degree of decomposition per requirement can be specified as a *PathLength* metric with respect to the *moreAbstractDescription* reference between requirements (cf. metamodel).

```
metric PathLength 18a_N is
elements satisfy "(Requirement.isInstance(self)"
references followed "moreAbstractDescription"
endmetric
```

5.4 Conclusion

By implementing the requirements metrics listed in Table 1 using the MDM approach, we are able to generate a requirements measurement tool. Concretely, it is an Eclipse plugin that measures models of requirements produced by EMF-based editors (Eclipse Modeling Framework, see [31]). Note that no code has been written, the only artifacts created were the requirements metamodel and the specifications of metrics within MDM.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

6 Discussion

This section discusses the important characteristics of our approach: the unified metric list, the metric-driven metamodeling process, the requirements metamodel, the implementation and the interpretation of metrics, and the main limitations.

6.1 Unified Metric List

The list of metrics of Table 1 is a consolidation of published work. We refer to the evaluation of each paper to assess the usefulness and applicability of metrics.

However, our contribution is not to define new metrics, but to present a unified list in order to show that it is possible to automate the production of requirements measurement software, and to contribute to build a consensus around requirements metrics in the requirements engineering research and industrial communities.

The previous metric frameworks suffered from two limitations: first, they addressed a particular facet of the requirements engineering process, and second they were not directly computable. On the contrary, our approach and our unified metric list aim to be comprehensive in scope (by unifying previous work). Furthermore, as discussed more in depth in what follows, this unified list grounds executable metric specifications (by defining a requirement metamodel and defining metrics as an instance of an executable metric metamodel).

A notable point is that this list contains few metrics related to non-functional requirements (NFRs). Metric #71 (Number of req. that describe architecture and algorithm) refers to them (see 2 for an explanation on how architectural constraints relate to NFRs). Metrics #57 and #70 (*Number of functional requirements allocated to a project release* and *Number of req. that describe pure external behavior*) focus on the functional requirements, hence indirectly relate to the dichotomy functional / non-functional requirements. Since this table reflects our survey (see 2), it shows that the literature has not focused so far on measurement related to non-functional requirements.

6.2 Metric-driven Metamodeling Process

The metric driven metamodeling process was appropriate to create our requirements metamodel. This process, which is not described in our previous work, was also useful in measuring models from other domains. For instance, in [32], we have applied the process to create new metamodels related to the simulation of maritime surveillance systems. Then, we were able to obtain simulation metrics in a model-driven manner (e.g. the number of boats that the system can detect). We choose to present this process in detail in this paper, because the need for measurement is the main motivation to adopt a metamodeling process for requirements (in order to address the limitations of natural language processing techniques). On the contrary, for maritime surveillance systems, it

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

Metamodel element	Metric IDs
Activity(c)	#47, #46, #50, #51, #74, #49, #48
Actor(c)	#36, #37, #34, #35, #50
ArchitecturalConstraint(c)	#71
Baseline(c)	#54, #78
Boundary(c)	#34, #35
CSCI(c)	#29, #23, #13
CapabilityRequirement(c)	#70, #57, #63, #62, #44, #27, #28
ConceptionLevel(c)	#10, #11, #12, #14, #15, #16, #17, #8, #9
Diagram(c)	#38, #41
EndUser(c)	#65
Flow(c)	#47, #75, #63, #49, #48
Goal(c)	#51, #52, #77
Individual(c)	#60
Realease(c)	#57
Requirement(c)	#55, #56, #69, #68, #19, #65, #67, #58, #60, #59, #21, #7, #23, #22, #2, #1, #29, #78
RequirementAddition(c)	#3
RequirementCategory(c)	#59
RequirementChange(c)	#6, #7, #43, #54
RequirementDeletion(c)	#5
RequirementModification(c)	#4
RequirementsBaseline(c)	#2
Responsible(c)	#55, #56
SequenceDiagram(c)	#32
StakeHolder(c)	#76, #52
State(c)	#45, #44
Status(c)	#19, #21, #25, #24, #30, #31, #33
Stimulus(c)	#62
TestCase(c)	#67, #68
TimeFrame(c)	#18, #78, #6, #4, #5, #25, #3
UseCase(c)	#47, #46, #45, #42, #41, #53, #40, #49, #48, #36, #26, #35, #25, #24, #74, #32, #38, #39
UseCaseDiagram(c)	#39, #33, #30, #31
allocatedTo(r)	#29, #23, #13
alternative(a)	#75
concrete(a)	#35
correct(a)	#65
costOfTest(a)	#67
decomposedIn(r)	#18, #43, #10, #11, #12, #13, #14, #15, #16, #17, #21, #20, #23, #29, #9
dependsOn(r)	#28, #58
describedBy(r)	#41
extends(r)	#40, #41, #53
includes(r)	#40, #42, #41, #53
next(r)	#74
originalRequirement(r)	#10, #11, #12, #14, #15, #16, #17, #8
reviewed(a)	#60
supplierFor(r)	#43, #58

Table 2: Main Metamodel Elements Supporting Requirements Metric

was as important to measure models and to simulate them.

6.3 Relationship between the Requirements Metamodel and the Metrics

To understand the relationship between our requirements metamodel and measurement, Table 2 shows the correspondence between the elements of the metamodel and the metrics in which they are involved. The first column gives the metamodel element (with a symbol to denote the kind of metamodel element: "c" for class, "r" for reference, "a" for attribute). The second column gives the metric ID (the ID refers to Table 1). For instance, the class "SequenceDiagram" (2nd column) is used in metric #32, which is the *Number of sequence diagrams per use case* (see Table 1). This table shows that some metamodel elements are very important for quantitatively assessing requirements: those elements that are involved in many metric formulas (i.e. the concept of

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

"UseCase"). Also, it shows that the metamodel covers all aspects of requirements engineering as given by the current state of research on requirements measurement.

The requirements metamodel not only supports metrics: with the Eclipse Modeling Framework [31], it is used to fully generate a requirements editor. A requirements editor supports the creation and modification of requirements specification in manner that complies with the structure enforced by the requirements metamodel.

6.4 Comparison with other Requirement Metamodels

A unique characteristic of the requirements metamodel proposed in this paper is the process through which it has been designed. This metamodel is designed as the set of necessary notions to compute all the requirement metrics we have found. This particular approach to metamodel design also means that the intention of the metamodel is unique: it is meant to formally capture requirements in a way that allows computing metrics, as opposed to modeling requirements for simulation or ambiguity detection.

Since this metamodel is designed from existing work on requirements, its content overlaps some existing metamodels. In particular concepts such as Requirement, UseCase or TestCase from our metamodel can be found in the requirements modeling part of SysML [33]. Still, since our intent focuses on reasoning and computing metrics about requirements and not about the system to which these requirements refer, we do not model the relations between requirements and design as it is done in SysML. Our metamodel also overlaps with the part of the UML metamodel dedicated to use cases (or similar use case metamodels such as [9]), but it is much more focused towards requirements engineering (for instance, we have the notion of requirement version). So, in summary, it overlaps with several existing metamodels (UML, SysML, REMM [34]), but it is the only metamodel that captures the concepts (and only the ones) that are necessary to compute all metrics from the litterature.

There are also requirements metamodels in the litterature that do not overlap with our metamodel. These metamodels either focus on one specific type of relationship among requirements (*e.g.*, the metamodel proposed by Gokni et al.[35]) or they go in the details of specific requirements such as real-time properties (*e.g.*, the work by Dhaussy et al. [36]) or detailed use case specifications (*e.g.*, the work by Brottier et al. [37]).

6.5 Implementation of Requirements Metrics

The computability of requirements metrics is the ability to automatically obtain metric values from requirements metric specifications. Contrary to previous work on requirements metrics, thanks to the MDM approach, we are able to create a formal and computable description of requirements metrics.

Also, the requirements measurement software is fully generated. Without any programming

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

effort, users get an integrated measurement tool in their requirements environment. For instance, a right click on a requirements document file proposes a "Measure" action, which computes the values of the 78 metrics listed of Table 1.

Since the whole code of the measurement tool is generated, the approach is adaptable. Both requirements metamodel and requirements metrics can be adapted or extended to a requirements engineering process specific to a company. For instance, the set of status for a requirement can be reduced or augmented, depending on the approval process. A company can also add a class to the metamodel, for instance a class *BudgetConstraint*, inheriting from a *Constraint* requirement (see Figure 3). Similarly, the requirements metric specifications can be adapted, and it is also possible to write new requirements metrics tailored to a given process.

6.6 Interpretation of Metric Values

The scope of the paper is: a requirements metamodel, a unification of the literature on requirements metrics and an automated approach for requirements measurement. Hence, it is out of scope here to provide interpretation guidelines of metric values. For this very important yet difficult point (the interpretation may depend on the company and project settings), we refer to both the papers that proposed the metrics, and to reference work in the domain [38, 39].

6.7 Drawbacks

We have shown above that using a model-driven approach for measuring requirements can provide a unified framework to formally express requirements and requirements metrics. However, this is no silver bullet. We identify two important drawbacks. First, requirements engineers have to change the way they think and produce requirements: they have to understand the requirements metamodel so as to fill the correct information as an instance of metamodel elements. They may also have to learn metamodeling to adapt the metamodel to their needs and to their existing processes, as discussed in section 6.5.

In industry, requirements engineers already use tools. Using our approach would introduce a new tool in their toolbox. This would introduce licensing costs, training costs and interoperability problems between tools. The latter point contains interesting areas of future research with respect to model interchange and requirements engineering processes.

7 Conclusion

In this paper, we have presented a new approach for the measurement of requirements. We analyzed 11 previous contributions on requirements metrics, consisting of 138 metric specifications. From this set of metrics, we have created a requirements metamodel and a consolidated list of 78 metric specifications. We have shown how to implement these metrics using the MDM approach

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

[16], a declarative and generative approach for measurement. Thanks to generative programming, our approach to requirements measurement allows to obtain both a requirements editor and a requirements measurement software.

Future work could explore whether it is possible to semi-automatically translate an existing requirements specification as a formalized specification that conforms to the proposed requirements metamodel. Also, an empirical study with practitioners would be valuable to highlight to which extent requirements processes and practices are open to structured requirements models.

References

[1] T. Hammer, L. Rosenberg, L. Huffman, and L. Hyatt, "Requirements metrics - value added," in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, p. 141.1, IEEE Computer Society, 1997.

[2] M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, and M. Bush, "Key practices of the capability maturity model," tech. rep., Software Engineering Institute, 1993.

[3] ISO/IEC, "Software product quality requirements and evaluation (square) (ISO/IEC 25000)," tech. rep., ISO/IEC, 2007.

[4] P. Zave, "Classification of research efforts in requirements engineering," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 315–321, 1997.

[5] A. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos, "Identifying and measuring quality in a software requirements specification," in *Proceedings of the First International Software Metrics Symposium*, IEEE, 1993.

[6] R. J. Costello and D.-B. Liu, "Metrics for requirements engineering," *J. Syst. Softw.*, vol. 29, pp. 39–63, Apr. 1995.

[7] M. Marchesi, "OOA metrics for the Unified Modeling Language," in *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, p. 67, IEEE Computer Society, 1998.

[8] A. Loconsole, "Measuring the requirements management key process area," in *Proceedings of the 12th European Software Control and Metrics Conference (ESCOM'2001)*, Shaker Publishing, 2001.

[9] B. Henderson-Sellers, D. Zowghi, T. Klemola, and S. Parasuram, "Sizing use cases: How to create a standard metrical approach," in *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS '02)*, pp. 409–421, Springer-Verlag, 2002.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

[10] Y. Singh, S. Sabharwal, and M. Sood, "A systematic approach to measure the problem complexity of software requirement specifications of an information system," *Information and Management Sciences*, vol. 15, pp. 69–90, 2004.

[11] B. Berenbach and G. Borotto, "Metrics for model driven requirements development," in *Proceeding of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 445–451, ACM Press, 2006.

[12] M. Medina Mora and C. Denger, "Requirements metrics: an initial literature survey on measurement approaches for requirements specifications," tech. rep., Fraunhofer IESE, 2003.

[13] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, pp. 25–31, February 2006.

[14] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Requirements by contracts allow automated system testing," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, p. 85, 2003.

[15] E. Brottier, B. Baudry, Y. L. Traon, D. Touzet, and B. Nicolas, "Producing a global requirement model from multiple requirement specifications," in *Proceedings of the IEEE Enterprise Computing Conference (EDOC'2007)*, pp. 390–404, 2007.

[16] M. Monperrus, J.-M. Jézéquel, J. Champeau, and B. Hoeltzener, "A model-driven measurement approach," in *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'2008)*, Springer, 2008.

[17] H. Zhang and M. A. Babar, "On searching relevant studies in software engineering," in *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2010.

[18] E. Neuhaus, C. Neuhaus, A. Asher, and C. Wrede, "The depth and breadth of Google Scholar: An empirical study," *Libraries and the Academy*, vol. 6, no. 2, 2006.

[19] X. Chen, "Google Scholar's Dramatic Coverage Improvement Five Years after Debut," *Serials Review*, 2010.

[20] IEEE, "Recommended practice for software requirements specifications (IEEE 830)," tech. rep., IEEE, 1998.

[21] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements," *Software: Practice and Experience*, vol. 32, no. 2, pp. 113–133, 2002.

[22] J. Baumert and M. McWhinney, "Software measures and the capability maturity model," tech. rep., Software Engineering Institute, Carnegie Mellon University, 1992.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

[23] OMG, "UML 2.0 superstructure," tech. rep., Object Management Group, 2004.

[24] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, Wiley, 1994.

[25] C. Kolde, "Basic metrics for requirements management." White paper, Borland, 2004.

[26] B. P. Douglass, "Computing model complexity." White paper, I-Logix, 2004.

[27] Modelware Project, "D2.2 MDD Engineering Metrics Definition," tech. rep., Framework Programme Information Society Technologies, 2006.

[28] R. Dromey, "Cornering the chimera," *IEEE Software*, vol. 13, no. 1, pp. 33–43, 1996.

[29] F. Moisiadis, "The fundamentals of prioritising requirements," in *Proceedings of the Systems Engineering, Test and Evaluation Conference (SETE'2002)*, The Systems Engineering Society of Australia, 2002.

[30] M. Monperrus, J.-M. Jézéquel, B. Baudry, J. Champeau, and B. Hoeltzener, "Model-driven generative development of measurement software," *Software and Systems Modeling (SoSyM)*, pp. –, 2010.

[31] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[32] M. Monperrus, B. Long, J. Champeau, B. Hoeltzener, G. Marchalot, and J.-M. Jézéquel, "Model-driven architecture of a maritime surveillance system simulator," *Systems Engineering Journal*, vol. 13, 2009.

[33] OMG, "Omg systems modeling language," Specification document 1.2, OMG, 2010.

[34] C. Vicente-Chicote, B. Moros, and J. A. T. Álvarez, "Remm-studio: an integrated model-driven environment for requirements specification, validation and formatting," *Journal of Object Technology*, vol. 6, no. 9, pp. 437–454, 2007.

[35] A. Goknil, I. Kurtev, and K. van den Berg, "A metamodeling approach for reasoning about requirements," in *Model Driven Architecture – Foundations and Applications* (I. Schieferdecker and A. Hartman, eds.), vol. 5095 of *Lecture Notes in Computer Science*, pp. 310–325, Springer Berlin / Heidelberg, 2008.

[36] P. Dhaussy, P.-Y. Pillain, S. Creff, A. Raji, Y. Le Traon, and B. Baudry, "Evaluating context descriptions and property definition patterns for software formal validation," in *Model Driven Engineering Languages and Systems* (A. Schürr and B. Selic, eds.), vol. 5795 of *Lecture Notes in Computer Science*, pp. 438–452, Springer Berlin / Heidelberg, 2009.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

[37] E. Brottier, Y. Le Traon, and B. Nicolas, "Composing models at two modeling levels to capture heterogeneous concerns in requirements," in *Software Composition* (B. Baudry and E. Wohlstadter, eds.), vol. 6144 of *Lecture Notes in Computer Science*, pp. 1–16, Springer Berlin / Heidelberg, 2010.

[38] R. Lutowski, *Software requirements: encapsulation, quality, and reuse*. Auerbach Publications, 2005.

[39] R. R. Young, *Project Requirements: A Guide to Best Practices*. Management Concepts, 2006.

In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

A The requirements metamodel

This description of the metamodel uses the syntax of the Kermeta metamodeling language, see `http://www.kermeta.org`

```
class Requirement {
        reference category : RequirementCategory#requirements
        reference level : ConceptionLevel#reqs
        reference moreAbstractDescription : Requirement[0..*]#refinedIn
        reference refinedIn : Requirement[0..*]#moreAbstractDescription
        reference status : Status[0..*]#reqs
        reference pastVersions : Requirement[0..*]#currentVersion
        reference currentVersion : Requirement#pastVersions
        attribute linkedMaterials : String[0..*]
        reference allocatedTo : CSCI[0..*]
        reference testCases : TestCase[0..*]
        reference responsible : Responsible[0..*]#requirements
        reference dependencies : Requirement[0..*]#providerFor
        reference providerFor : Requirement[0..*]#dependencies
        attribute name : String
        attribute description : String
        attribute creationDate : Date
        attribute author : String }
class RequirementCategory {
        reference requirements : Requirement[0..*]#category
        attribute name : String }
class UseCase inherits CapabilityRequirement {
        reference diagrams : UseCaseDiagram[0..*]#usecases
        reference includes : UseCase[0..*]
        attribute ~abstract : boolean
        reference describedBy : DynamicDiagram#describedUseCase
        reference extends : UseCase[0..*] }
class RequirementAddition inherits RequirementChange {
        reference newRequirement : Requirement[1..1] }
class RequirementDeletion inherits RequirementChange {
        reference deletedRequirement : Requirement[1..1] }
class RequirementModification inherits RequirementChange {
        reference newVersion : Requirement[1..1]
```


In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

```
        reference oldVersion : Requirement[1..1] }
class RequirementChange {
        attribute date : String }
class ConceptionLevel {
        reference reqs : Requirement[0..*]#level
        attribute num : integer
        attribute description : String
        reference nextLevel : ConceptionLevel }
class Status {
        reference reqs : Requirement[0..*]#status
        attribute name : String }
class UseCaseDiagram {
        reference usecases : UseCase[0..*]#diagrams
        reference status : Status[0..*] }
class TimeFrame {
        reference changes : RequirementChange[0..*] }
class CapabilityRequirement inherits Requirement {
        reference actors : Actor[0..*]#inreq }
class ConstraintRequirement inherits Requirement { }
class ArchitecturalConstraint inherits ConstraintRequirement { }
class PerformanceConstraint inherits ConstraintRequirement { }
class Actor {
        reference inreq : CapabilityRequirement[0..*]#actors
        reference inUseCase : UseCase[0..*]
        reference boundary : Boundary[0..*]#actor
        attribute complexity : integer }
class CSCI {
        attribute description : String }
class Class inherits CSCI { }
class Procedure inherits CSCI { }
class Method inherits CSCI { }
class Boundary {
        reference actor : Actor[0..*]#boundary }
class DynamicDiagram {
        reference describedUseCase : UseCase#describedBy }
class StateDiagram inherits DynamicDiagram {
        attribute states : State[0..*]#containingStateDiagram
```


In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)

```
        reference stimuli : Stimulus[0..*] }
class SequenceDiagram inherits DynamicDiagram { }
class ActivityDiagram inherits DynamicDiagram {
        attribute activities : Activity[0..*]#containingActivityDiagram }
class Activity {
        reference containingActivityDiagram : ActivityDiagram#activities }
class State {
        reference containingStateDiagram : StateDiagram#states }
class Defects {
        reference faultyCSCI : CSCI[0..*] }
class ProjectRelease {
        reference requirementsSatisfied : Requirement[0..*] }
class Function inherits CapabilityRequirement { }
class Stimulus {
        reference usedIn : StateDiagram[0..*] }
class TestCase {
        attribute estimatedDevelopmentCostOfTestCase : integer
        attribute estimatedTimeCostOfTestCase : integer
        attribute realDevelopmentCostOfTestCase : integer
        attribute realDevelopmentTimeOfTestCase : integer }
class RequirementsBaseLine {
        reference requirements : Requirement[0..*] }
class Responsible {
        reference requirements : Requirement[0..*]#responsible }
class Group inherits Responsible {
        reference individuals : Individual[0..*]#groups }
class Individual inherits Responsible {
        reference groups : Group[0..*]#individuals }
class CSCIStatus inherits Status { }
```


In: Software Quality Journal, Springer, Online Edition, 2011 (DOI: 10.1007/s11219-011-9163-6)